REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Dec 93	Technical

4. TITLE AND SUBTITLE
Parallel Adaptive Mesh Refinement and Redistribution on Distributed Memory Computers

5. FUNDING NUMBERS
DAAL03-91-G-0215

6. AUTHOR(S)
C. Ozturan, H.L. deCougny, M.S. Shephard, J.E. Flaherty

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Rensselaer Polytechnic Institute
Troy, New York 12180-3590

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U.S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
ARO 29167.24-MA

11. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

ABSTRACT ON FIRST PAGE OF REPORT

14. SUBJECT TERMS

15. NUMBER OF PAGES
18

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Rensselaer

Department of Computer Science
Technical Report

Parallel Adaptive Mesh Refinement and Redistribution on Distributed Memory Computers

C. Özturan, H.L. deCougny, M.S. Shephard, J.E. Flaherty
Scientific Computation Research Center

Rensselaer Polytechnic Institute
Troy, New York 12180-3590

Report No. 93-26 **December 1993**

Parallel Adaptive Mesh Refinement and Redistribution on Distributed Memory Computers

C. Özturan, H. L. deCougny, M. S. Shephard, J.E. Flaherty
Scientific Computation Research Center
Rensselaer Polytechnic Institute
Troy, NY 12181

Abstract

A procedure to support parallel refinement and redistribution of two dimensional unstructured finite element meshes on distributed memory computers is presented. The procedure uses the mesh topological entity hierarchy as the underlying data structures to easily support the required adjacency information. Mesh refinement is done by employing links back to the geometric representation to place new nodes on the boundary of the domain directly on the curved geometry. The refined mesh is then redistributed by an iterative heuristic based on the Leiss/Reddy [9] load balancing criteria. A fast parallel tree edge-coloring algorithm is used to pair processors having adjacent partitions and forming a tree structure as a result of Leiss/Reddy load request criteria. Excess elements are iteratively migrated from heavily loaded to less loaded processors until load balancing is achieved. The system is implemented on a massively parallel MasPar MP-1 system with a SIMD style of computation and uses message passing primitives to migrate elements during the mesh redistribution phase. Performance results of the redistribution heuristics on various test meshes are given.

1 Introduction

Adaptive finite element methods, driven by automatic estimation and control of errors have gained importance recently due to their ability to offer reliable solutions to partial differential equations [6]. An adaptive method starts with a solution on a coarse mesh using a low-order method and, based on an estimate of the global and local errors, either refines the mesh (*h-refinement*) and/or increases the order of numerical solution (*p-refinement*). The sequential implementations of these methods have proved to be very efficient by concentrating computations on regions of high activity and by providing exponential rates of convergence when proper combinations of h- and p-refinement are employed.

The irregular and evolving behavior of the computational load in adaptive strategies on complex domains becomes problematic when parallel distributed-memory machine implementations are considered. Complete parallelizations of these methods necessitate additional and difficult stages of partitioning, parallel refinement and the redistribution of the refined mesh. Many heuristics have been devised to partition the initial unstructured mesh and hence minimize the load imbalance and interprocessor communication among processors. The redistribution of the refined mesh can also be done by parallelizing similar partitioning heuristics. We note, however, that global methods such as recursive mesh subdivision techniques which operate on the whole mesh do not take effective advantage of the incremental changes in the refined mesh. Changes in the mesh might not call for redistribution because the cost of redistributing may be more than the cost of performing the computations with a slightly imbalanced load. More importantly, the redistribution may involve

only local adjustments. Therefore the use of heuristics that operate locally by migrating elements between the mapped neighboring partitions is an attractive alternative.

This paper presents a procedure for parallel refinement and redistribution of two dimensional finite element meshes on distributed-memory computers. Section 2, reviews the currently available partitioning and redistribution techniques and discusses the suitability of these techniques for adaptive techniques. Sections 3 and 4 present the data structures that are used to store the mesh, the adjacency links between the processors and the specific geometric operators needed to handle h-refinement on curved boundaries. The refined mesh is redistributed by employing an iterative heuristic based on taking pairs of processors having adjacent partitions and migrating elements from heavily loaded to less loaded processor. We describe the redistribution algorithm in Section 5. Finally, we report performance results of the redistribution heuristic based on SIMD style of computation and message passing primitives on a 2048-processor massively parallel MasPar MP-1 system.

2 Related Work

The current repertoire of partitioning and redistribution algorithms can be classified into three categories.

1. *Recursive Bisection (RB) Techniques* which repeatedly split the mesh into two-submeshes. *Coordinate RB* methods bisect the elements by their spatial coordinates. If the axis of bisection is Cartesian, then it is called *Orthogonal RB* [2]. If the axes are chosen to be along the principal axis of the moment of inertia matrix, then it is called *Moment RB. Spectral RB* is another method which utilizes the properties of the *Laplacian matrix* [5] of the mesh connectivity graph and bisects it according to the eigenvector corresponding to the second smallest eigenvalue of this matrix [11].

2. *Probabilistic Methods* which include simulated annealing and genetic algorithms. These methods however require many iterations and are expensive to use as mesh partitioning methods [20].

3. *Iterative Local Migration Techniques* exchange load between neighboring processors to improve the load balance and/or decrease the communication volume. The definition of processor neighborhood can either be the hardware link or the connectivity of the split domains. The *Cyclic pairwise exchange* [7] pairs processors connected by a hardware link and exchanges the nodes of the mesh to improve the communication. Leiss/Reddy on the other hand uses the hardware link as the neighborhood to transfer work from heavily loaded to less loaded processors. The *Tiling* algorithm [4][19] extends the Leiss/Reddy algorithm to the case where the neighborhood is defined by the connectivity of the split domains. Lohner et al. [10] algorithm exchange elements between subdomains according to a deficit difference function which reflects the imbalance between an element and its neighbors.

Adaptive methods refine the mesh incrementally and hence require periodic redistribution of finite elements. If used on the whole mesh, RB methods require complete remapping of the elements and therefore have a fixed cost. This introduces substantial overhead due to transfer of all elements to their new destinations. RB methods can handle the heterogenous load distributions on each element by assigning weights to each element. Recent efforts incorporate the communication costs into the ORB method by considering the weights on the cut edges of the split domains [3]. RB methods applied *globally* on the whole mesh are too costly to be used repeatedly to redistribute a

mesh dynamically with an adaptive technique. They may however, offer their advantages if applied *locally* to the incrementally altered mesh [16].

Iterative local migration techniques offer important advantages with an adaptive technique. Since these methods perform local transfers, incremental changes in the mesh can be propagated to the processors to load balance and reduce communication volume without solving an expensive global partitioning problem. The elemental cost, which is the sum of the computational workload represented by the degrees of freedom (dofs) and the communication, represented by the dofs on the partition boundary edges, can also be handled by selecting elements according to a cost function for transfer. A disadvantage of iterative local migration techniques is that many iterations may be required to regain global balance and hence elements reach their final destination after many local transfers rather than directly. Assuming each iteration is fast the cost due to local transfer steps is amortized by the smaller number of elements that are moved. Hence, they should be advantageous for use with adaptive techniques.

The redistribution strategy given here is an iterative local migration scheme. It is based on the Leiss/Reddy [9] algorithm and employs selection criteria similar to Wheat et al. [19] in transferring elements. Unlike, these approaches, however, the processors are paired during load transfers similar to the pairwise exchange heuristic used by Hammond [7]. Our pairing procedure does not pair processors connected by hardware link in the static processor graph, but rather in the dynamically changing graph representing the partitioned mesh (see Section 5).

3 Mesh Data Structures

Parallel h- and p-refinement and dynamic redistribution algorithms for the refined mesh require various mesh adjacency and mesh entity classification information. The current procedure operates on the triangular unstructured meshes generated by the shell capability of the *Finite Octree* procedure [13]. The data structure used in this mesh generator is the complete mesh topological *entity hierarchy* [18] which provide a two-way link between the mesh entities of consecutive order, i.e., regions, faces, edges, and vertices. Although this hierarchical data structure requires more memory than classic finite element data structures (e.g., element-node relationship), they have proven to be powerful especially in the context of refinement. It is quite clear that h-refinement benefits from a complete hierarchy to delete and create mesh entities efficiently. The presence of a complete hierarchy is also very useful with the p-version of the finite element method, since it easy to attach the edge and interior modes [14] to the mesh entities. Additionally, mesh entities are explicitly classified against the geometric model describing the domain boundary. In two dimensions, mesh faces are always classified as being in the interior of the domain, mesh edges are classified either in the interior or on the boundary edge, and mesh vertices are classified either in the interior, on boundary edge, or on a boundary vertex. Classification guarantees that the mesh is still valid after h-refinement. Also, any mesh vertex classified on a boundary edge stores its parametric coordinate on the corresponding boundary edge. Since generally large elements are used in the p-version, it is important to represent edges on curved boundaries accurately. The readily available classification and the parametric coordinates make it easy to implement special boundary-element mapping techniques. Figure 1(a) illustrates the relationship between the entities and the geometric model in the data structures.

Because mesh entities are distributed across processors corresponding to the current partitioning, additional data has to be stored in order for a partition to have access to information stored on neighboring partition. In two dimensions, each mesh edge on a partition's boundary points to a corresponding identical, duplicate mesh edge on the neighboring partition's boundary. Storing

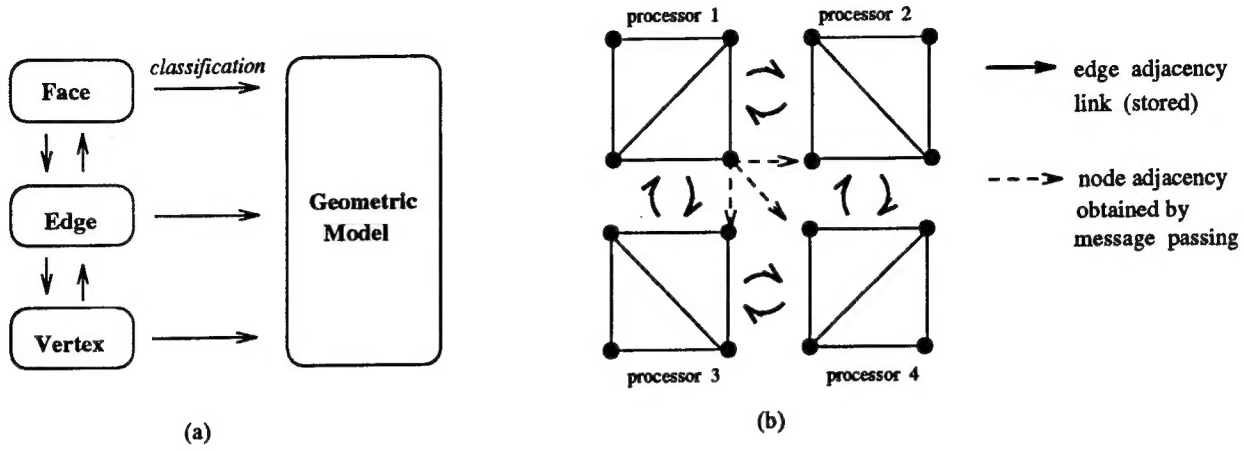

Figure 1: *(a) Mesh data structures (b) across processor links*

links between these duplicate edges is sufficient to allow for inter-processor mesh entity access. The additional information regarding inter-processor vertex connectivities is obtained by sending messages and following the edge links across processors. Although this approach introduces additional communication, it saves substantial amount of memory by alleviating the problem of storing variable length vertex connectivity lists. Figure 1(b) shows an example of stored edge links across partition boundaries.

4 Parallel H-refinement

The h-refinement procedure is edge-based; thus each mesh edge in the mesh is either marked for refinement or not. Duplicate mesh edges on partition boundaries must be marked identically. Each mesh face can have from zero to three marked mesh edges. For each possible configuration, a template has been defined [17] as shown in Figure 2:

1. 1-edge: the triangle is divided into two (also known as a *green* subdivision [1]).
2. 2-edge: the triangle is divided into three. In this case, there is always a choice between two subdivisions. In order to limit element quality degradation, the cut along longest edge is performed first [12].
3. 3-edge: the triangle is subdivided into four. For simplicity, the subdivision that produces 4 triangles similar to the parent is chosen (also known as a *regular* subdivision [1]).

Elements are subdivided by traversing the list of mesh faces known by the processor in parallel. Once all mesh faces have been visited, mesh edges resulting from refinement on a partition boundary are linked to their identical duplicates on the neighboring partition. The refined parent entities are not deleted immediately and separate operators are provided for this purpose. In this way, information can be transferred from the parent entities to their off-spring before the deletion.

Any mesh vertex or edge resulting from the refinement of a mesh edge inherits the classification of the parent edge. With curved geometry, refinement vertices that are classified on the model boundary are snapped to the proper model entity. The parametric coordinate of any refinement vertex classified on the model boundary is obtained as a simple average of the parametric coordinates

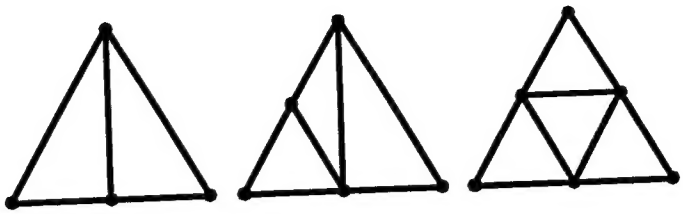

Figure 2: *Refinement templates.*

of the two end vertices of the parent edge. The location of the refinement vertex is then obtained by querying the geometric modeler. It is possible that the location of the refinement vertex induces negative jacobians for the surrounding elements. In this case, one has to either: (*i*) slightly move the location of the refinement vertex, (*ii*) remove the elements with negative jacobians, and/or (*iii*) add an additional vertex.

5 Mesh Redistribution

The redistribution algorithm and its similarities and differences with other implementations [9][4][19] are described using an example. Consider the unbalanced mesh distribution over nine processors as shown in Figure 3(a). Let $G_P(V, E)$ be a *partition graph* with each vertex in V representing a partition assigned to a processor and E representing the set of edges between partitions. Two partitions u and v are connected by an edge $(u, v) \in E$ if they share a mesh edge. If two partitions share only a mesh vertex, then they are not considered adjacent in the partition graph. The reason for the mesh edge connectivity requirement between partitions in G_P is twofold. First, by excluding vertex adjacency, the number of edges in E and, hence, the time to communicate with adjacent processors is kept minimal. Second, transferring elements to a partition which share only a node grows a partition with two regions connected only by a single vertex. This, in turn, results in a higher surface to volume ratio which increases communication cost. Figure 3(b) shows the partition graph obtained from the mesh distribution in 3(a).

Following Leiss/Reddy [9], a workload deficient processor will request work from its most heavily loaded neighbor. As a result, a processor can receive multiple requests but can only request load from one processor. This pattern of requests produces a load hierarchy and forms a forest of trees T_i as shown in Figure 3(c). The trees T_i in the forest are subgraphs of the partition graph G_P.

The current proposed algorithm for redistribution pairs the processors on each tree T_i and transfers load from the heavily loaded pair to the other. The pairing of processors is equivalent to coloring the edges of each tree T_i with colors representing separate load transfer (communication) cycles. The edge coloring approach synchronizes the load transfer between neighboring processors and differs from the approach of Wheat et al. [19]. According to our implementation, processors P_2 and P_7 in Figure 3(b) would be engaged with load transfer with only one neighbor at a single transfer step. Wheat et al. [19], however, would let processor P_2 and P_7 receive and send work during the same transfer step. The latter approach would be more efficient if there were load transfer in one direction only, i.e. from heavily loaded to less loaded processor. Processors P_2 and P_7 would be doing useful work packing elements to be transferred to their offspring while their parents pack elements to be transferred to them. In the current implementation, processors P_2 and P_7 would remain idle waiting for load transfer from parents and would transfer load to offspring after the transfer from the parent has been completed. However, this disadvantage is overshadowed

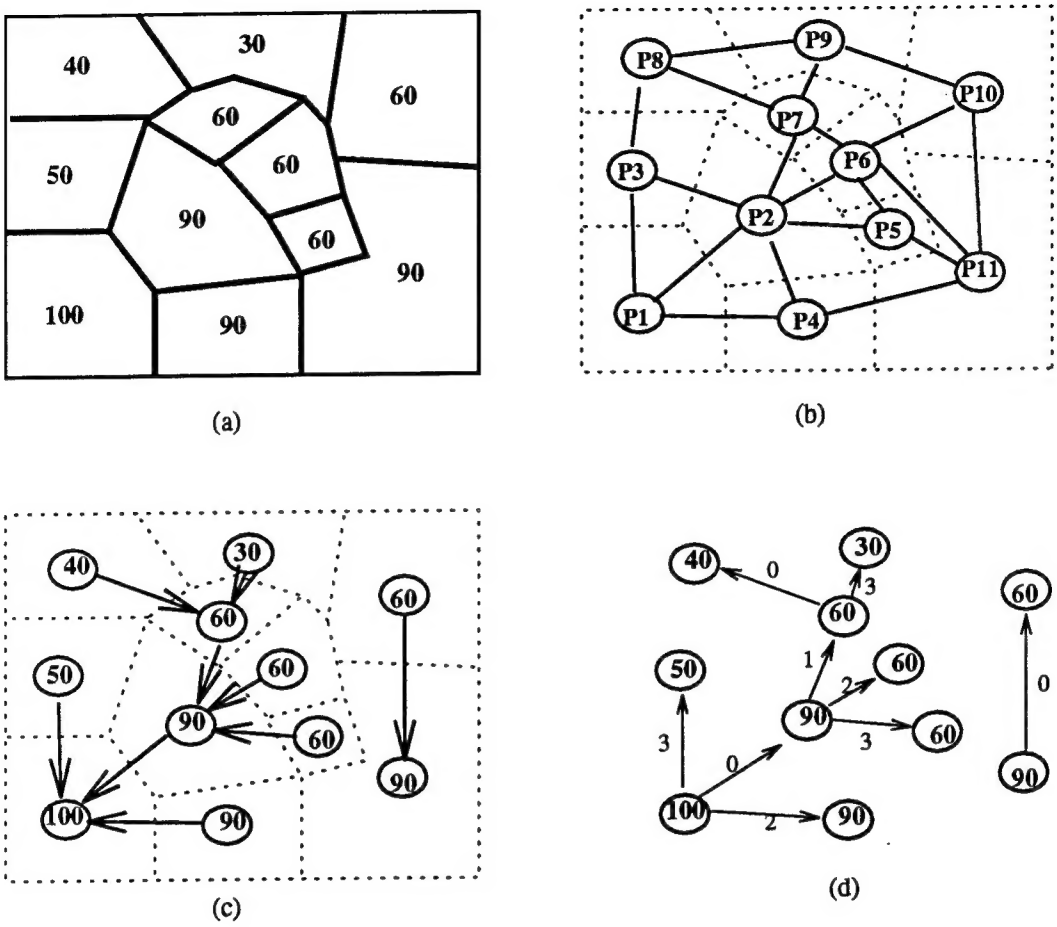

Figure 3: *(a) Unbalanced load in each partition, (b) partition graph P_G, (c) load request, (d) load transfer between pairs in steps 0,1,2,3.*

by a number of advantages. Firstly, smaller number of messages and the synchronous transfer of load increase communication performance. Second, since the processors are synchronized by pairs, a greater repertoire of selection criteria to decide which elements to transfer can be employed. Unlike [19], where elements can be transferred from only heavily load to less loaded processors, the pairing allows elements to be transferred from the less loaded to the heavily loaded pair. This can be useful in improving the surface to volume ratio of the partitions. Since there is no explicit synchronization by edge coloring in [19], this approach would make implementation of this bidirectional transfer of load extremely difficult.

Figure 3(d) shows the coloring phase that we employ to pair the processors. If $\Delta(G)$ denotes the maximum vertex degree (number of edges incident on a vertex) in a graph G, then Vizing's theorem [15] shows that the graph G can be edge-colored using C colors where $\Delta(G) \leq C \leq \Delta(G)+1$. For some special graphs including the trees the number of colors needed is exactly $\Delta(G)$. Therefore, $\Delta(T_i)$ colors are required to color the tree T_i. The main steps of the `REDISTRIBUTE` algorithm are illustrated in Figure 4.
Before presenting the details of REDISTRIBUTE, a summary of the steps is given.

- Step 1: The steps of transferring work between paired processors is iterated until the load

Algorithm REDISTRIBUTE:

Input: An initial mesh distributed on the processors.

Output: Redistributed mesh with reduced imbalance.

1. **While** not converged **do**
2. Compute neighboring load differences.
3. Request load from neighbor processor having largest load difference. (creates processor tree T)
4. Determine amount of load to be sent or received.
5. Set-up Euler tour adjacency links for tree T.
6. Color the tree T by Euler touring.

7. **for each** color C **do**
8. **if** processor owns color C **and** is a neighbor of color C pair processor **then**
9. Transfer load between the pair processors.

end for

end while

Figure 4: *Redistribution algorithm.*

on each processor converges to a value close to the optimal balance. The convergence of the Leiss/Reddy algorithm and the current algorithm's convergence criteria is given in Section 5.3.

- Step 2: Load differences are computed by having each processor send a load value to its neighbors and correspondingly receive load values from its neighbors. This step takes $\Delta(G_P)$ time.

- Step 3: The Leiss/Reddy [9] load request process is invoked and results in the forest of trees T_i. The edges of G_p is simply marked when a request has been made indicating whether or not it is a tree edge. Since the incoming requests for load should be sorted, this step takes $O(\max_i\{d_i \cdot \log d_i\})$ time where $d_i = \Delta(T_i)$.

- Step 4: Deciding how much load to transfer to requesting processors is crucial in making the redistribution algorithm to converge. Criteria for this step are given with convergence criteria in Section 5.3.

- Steps 5–6: To facilitate efficient parallel scan operations on the trees T_i, each tree is linearized by constructing an *Euler Tour* on it. The details of the Euler Tour construction can be found in [8]. In the present implementation, each processor stores the links to its adjacent processors; hence, this list is traversed and local addresses sent to the adjacent processors. The received messages are sorted and the links stored. Hence, constructing Euler Tour takes $O(\max_i\{d_i \cdot \log d_i\})$ time. Having constructed the Euler Tour, the tree is colored by employing scan operation on the linearized tree. The complexity of coloring is $O(\max_i\{d_i \cdot \log |V_i|\})$ where $|V_i|$ denotes the number of vertices in tree T_i. The details of tree edge coloring are given in Section 5.1.

- Steps 7 – 8: The steps of load transfer are synchronized by the edge coloring of the tree. One iteration of the redistribution algorithm involves C steps corresponding to the $\max_i(\Delta(T_i))$ colors. Note that this synchronization also allows for bi-directional transfer of load between pair processors.

- Step 9 selects elements to be transferred. A cost function is associated with each element to be transferred. This cost reflects the communication as well as the computational cost of the element. The details of this procedure are given in Section 5.2.

5.1 Fast Tree Edge-Coloring Algorithm to Pair Processors

Given a tree $T(V, E)$ with vertices V and edges E, an Euler Tour, $T_{Euler}(V, E')$, can be constructed by replacing each edge $(u, v) \in E$ by two directed arcs $< u, v > \in E'$ and $< v, u > \in E'$ and setting up Euler Tour adjacency links. The directed graph $T_{Euler}(V, E')$ linearizes the tree T and allows efficient global scan operations to be done by performing parallel pointer jumping. The edge coloring of the tree is performed by assigning special weights to the arcs in T_{Euler}, scan summing these weights and taking modulo the maximum tree vertex degree, $\Delta(T)$, of the summed weights.

Let $parent(u)$ denote the parent vertex of u in the rooted tree T, $num\_child(u)$ denote the number of children of v, and $child\_rank(u)$ denote the left to right rank of children of a parent vertex. Hence, if u is a parent vertex, then its leftmost child has $child\_rank = 1$ and its rightmost child has $child\_rank = num\_child(u)$. This classifies the arcs of T_{Euler} into three types, i.e., $< u, v >$ is:

- a *forward arc* if $u = parent(v)$.
- an *interior backward arc* if $v = parent(u)$ and $child\_rank(u) < num\_child(v)$
- a *last backward arc* if $v = parent(u)$ and $child\_rank(u) = num\_child(v)$

With these definitions, the tree edge-coloring algorithm is illustrated in Figure 5.

Tree Edge Coloring Algorithm:
Input: A tree T with an Euler Tour T_{Euler} defined on it.
Output: Edge coloring of the tree on forward edges.

1. Assign weights to each arc $< u, v >$:
   - weight=1 if forward arc.
   - weight=0 if interior backward arc.
   - weight= $\Delta(T) - num\_child(v)$ if last backward arc.
2. Scan sum weights by parallel pointer jumping.
3. Take summed weight modulo $\Delta(T)$ on forward arcs.

Figure 5: *Coloring Algorithm.*

Step 1 of the above algorithm takes $O(\Delta(T))$ time since each processor has to traverse its list of neighbors to make weight assignment. The maximum number of neighbors is given by the maximum degree $\Delta(T)$. Step 2 involves efficient parallel scan operation. Scan operations take logarithmic time if each processor stores only one item to be scanned. In the present case, each processor stores $\Delta(T)$ items and hence this list must be traversed during each step of the scanning.

Hence step 2 takes $O(\Delta(T) \cdot \log|V|)$. The complexity of step 3 is the same as step 1. Hence the overall coloring algorithm has $O(\Delta(T) \cdot \log|V|)$ parallel complexity for each tree. Figure 6 shows the coloring algorithm applied to one of the trees in the load redistribution example. Figure 6(a) shows the weight assignment on the tree, (b) shows the three steps of the algorithm on the linearized tree and (c) shows the resulting coloring assignment on the edges of the tree.

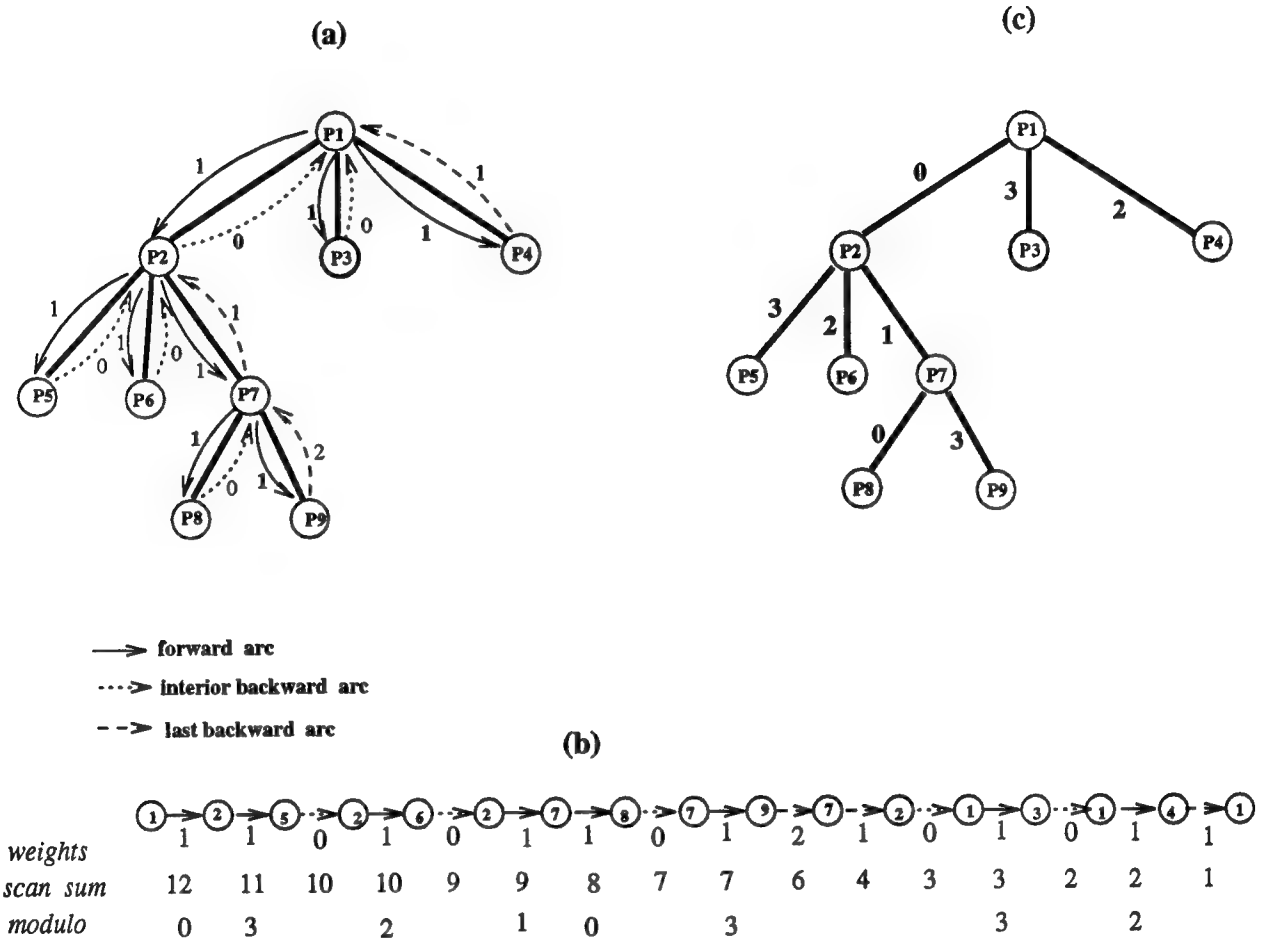

Figure 6: *Example (a) Weight assignment (b) Algorithm steps (c) Edge coloring obtained.*

Correctness of the Algorithm

An initial lemma is stated about the summation in a subtree of the weights assigned by the coloring algorithm and then this lemma is used to establish the correctness of the coloring algorithm.

Let set E'_f consist of the forward edges, E'_i the interior backward edges and E'_l, the last backward edges. Also let the non-leaf vertices of T be given by $R(T) = \{v | v \in V \ \ and \ \ num\_child(v) \neq 0\}$.

Lemma 1 *Given an Euler tour $T_{Euler}(V, E')$ of a rooted tree T, an integer constant $c \geq \Delta(T)$ and the assignment of weights to the arcs of T_{Euler}*

$$w(e) = \begin{cases} 1 & \text{if } e \in E'_f \\ 0 & \text{if } e \in E'_i \\ \textit{c-num\_child(v)} & \text{if } e \in E'_l \end{cases}$$

then

$$\sum_{e \in E'} w(e) = c \cdot |R(T)| \tag{1}$$

Proof: Simply summing the weights of all the edges gives:

$$\begin{aligned} \sum_{e \in E'} w(e) &= \sum_{e \in E'_f} w(e) + \sum_{e \in E'_i} w(e) + \sum_{e \in E'_l} w(e) \\ &= |V| - 1 + \sum_{e \in E'_l} (c - \textit{num\_child}(v)) \\ &= |V| - 1 + c \cdot |R(T)| - \sum_{e \in E'_l} \textit{num\_child}(v) \\ &= c \cdot |R(T)| \end{aligned}$$

□

We establish the correctness of the algorithm with the following theorem.

Theorem 1 The algorithm given in Figure 5 edge-colors a tree T using $\Delta(T)$ colors.

Proof: Note that each of the edges incident on a vertex should have a different number assigned to it. It suffices to examine the non-leaf vertices. Let the summed weights be denoted by w'. Given $u \in R(T)$, there are forward arcs to the children $< u, v_i > i = 1, \ldots, \textit{num\_child}(u)$. If u is not the root of the tree, the forward arc from the parent $< \textit{parent}(u), u >$. It remains to be shown that the summed weight modulo Δ is different for each of these arcs. First this is shown for the arcs $< u, v_i > i = 1, \ldots, \textit{num\_child}(u)$.

Let the $w'(< v_{\textit{num\_child}(u)}, u >) = s$. Then the summed weights for children are given as:

$$w'(u, v_i) = s + \textit{num\_child}(u) - i + 1 + \sum_{j=i}^{\textit{num\_child}(u)} \Delta(T) \cdot |R(T_i)| \qquad i = 1, \ldots, \textit{num\_child}(u)$$

where T_i denotes the subtree rooted by v_i and the last term is obtained using the Lemma. It follows that each edge color is given by:

$$\textit{color}(< u, v_i >) = w'(< u, v_i >) \textbf{ mod } \Delta(T) = (s + \textit{num\_child}(u) - i + 1) \textbf{ mod } \Delta(T)$$

There are two cases: 1) $\textit{num\_child}(u) = \Delta(T)$ which can occur if u is the root of T and $\textit{degree}(u) = \Delta(T)$, and 2) $\textit{num\_child}(u) < \Delta(T)$. In the first case, there is no $< \textit{parent}(u), u >$ arc, so $(s + \textit{num\_child}(u) - i + 1)$ **mod** Δ for $i = 1, \ldots, \Delta$ is distinct for the forward arcs. In the second case, u can have a parent or be the root in which case $(s + \textit{num\_child}(u) - i + 1)$ **mod** Δ for $1 \leq i \leq \textit{num\_child}(u) < \Delta$ which is distinct for all $< u, v_i >$. If u has a parent i.e. there is an arc $< \textit{parent}(u), u >$, then this edge gets $(s + \textit{num\_child}(u) + 1)$ **mod** Δ as the color which is distinct from the children's color. Thus since forward arcs dictate the color, then all the edges incident on a vertex have distinct colors.

5.2 Criteria for Selecting Elements to Transfer

Since the work per element can vary with adaptive methods, the selection criteria for deciding which elements to move becomes more complex. Consider one of the element movement criteria proposed by Lohner [10] as shown in Figure 7. To prevent noisy partition boundaries in (b), elements surrounding one of the vertices of the boundary edge are transferred as shown in (c). This approach however may pose problems with p-refinement. Since the degrees of freedom on the edges and/or in the interior of an element can be increased by p-refinement, the work as well as the communication requirements becomes spatially nonuniform. Hence, choosing elements around a boundary vertex is not always be the best criteria. Figure 7 (d) illustrates an example in which some of the elements have two degrees of freedom on their edges in addition to their nodal support. In this case selecting the elements which will cause noisy boundaries will be a better choice, because it decreases the communication volume. Figures 7(e) and (f) show the total number of degrees of freedom on the partition boundary in each case.

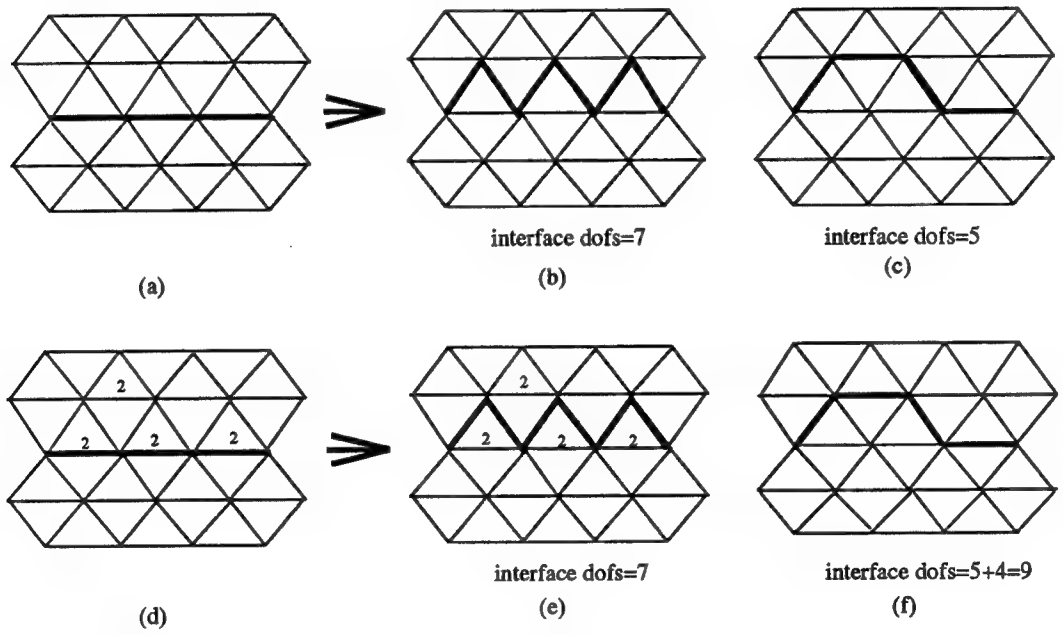

Figure 7: *Lohner's[10] example and selection criteria (a)-(c). The case when varying number of degrees of freedom are present on the edges (d)-(f).*

The most important criteria for element selection are:

1. Element selection should be done fast since it will be called for each element to be moved. A traversal of the list of partition boundary edges to locate the next element to move is required.
2. As the example of Figure 7 illustrates, the number of degrees of freedom on the partition boundary should be kept minimal. This should be done to reduce communication volume.
3. Elements with a larger load should be favored since this will imply fewer element transfers and faster convergence to load balance.

The current criteria for selecting elements is similar to the approach in Wheat et al. [19] which prioritizes the boundary elements. Each partition boundary element has a cost associated with it

based on:

1. The element workload which is the total number of degrees of freedom in the interior, on the edges and the nodes of the element.

2. The difference in communication cost that will result if the element is moved. This is equal to the total number of degrees of freedom on the edges and the nodes that will be exposed to the partition boundary.

3. The previous moves. This can be optionally used to favor selecting elements which are adjacent to a previously moved element.

The above three pieces of information are represented as a $3-tuple$ (L_e, D_e, P_e) and used as a cost indicator when moving the elements. The parameter L_e refers to the workload on the element and it is the most significant part of the cost. The higher L_e, the higher the priority of the element to move. The reason for assigning the heaviest weight to the workload on the element is to facilitate greater reduction in the imbalance between the processors when an element is moved. If all the elements have equal work, L_e is set to unity. In this case, the number of elements is balanced between the partitions. The parameter D_e refers to the difference in the communication cost and is the second most significant part. The communication volume might decrease in which case $D_e > 0$ or remain the same $D_e = 0$ or increase $D_e < 0$ when the element is moved. Finally, the least significant P_e part can be used as a history mechanism for the previous moves. When an element is moved, the P_e field of neighboring elements can be marked by 1 so that when the next element is to be moved, and both L_e and D_e are the same, the P_e can be used to break ties. In this case, since all parameters are equal, the marked P_e will enable the element which was the neighbor of a previously moved element to be favored. Given this cost assignment, the next element to move is then chosen as the one having the maximum (L_e, D_e, P_e) cost. We illustrate the use of 3-tuple cost template by applying it to the example mesh in Figure 8(a-c). For simplicity it is assumed that the number of edges gives the communication cost and each element has a workload value of 1. As a result boundary elements 2, 4, and 6 which are candidates for moving all have cost $(1, -1, 0)$. In this case, the load is $L_e = 1$. The communication cost is given by $D_e = 1 - 2$ which is the difference between the number of currently exposed edges to the number if the element were moved. Finally P_e is set to 0 indicating no neighbor has been moved yet. Since all costs are equal, an element will be chosen arbitrarily, which is taken to be element 2. The newly exposed elements 1 and 3 are inserted into the partition boundary element list and their costs calculated. The elements 1 and 3 also have their P_e marked as 1 since they are neighbor of a moved element. This field is then used to favor their movement in case ties arise in the other L_e and D_e costs. This process is repeated until all three elements have been moved.

5.3 Determining Amount of Load to Send and Convergence

Suppose a parent processor with load value L_0 has m load requesting offspring with load values L_i $i = 1, \ldots, m$ as shown in Figure 9(a). The criteria for determining how much load to transfer is as follows: Each child requests an amount r_i which is equal to the difference from its current load to the average of its and its parent's load, i.e.

$$r_i = \lceil (L_0 - L_i)/2 \rceil. \tag{2}$$

The parent processor will decide to send a total amount which will make its load become the average of the loads L_i $i = 0, \ldots, m$:

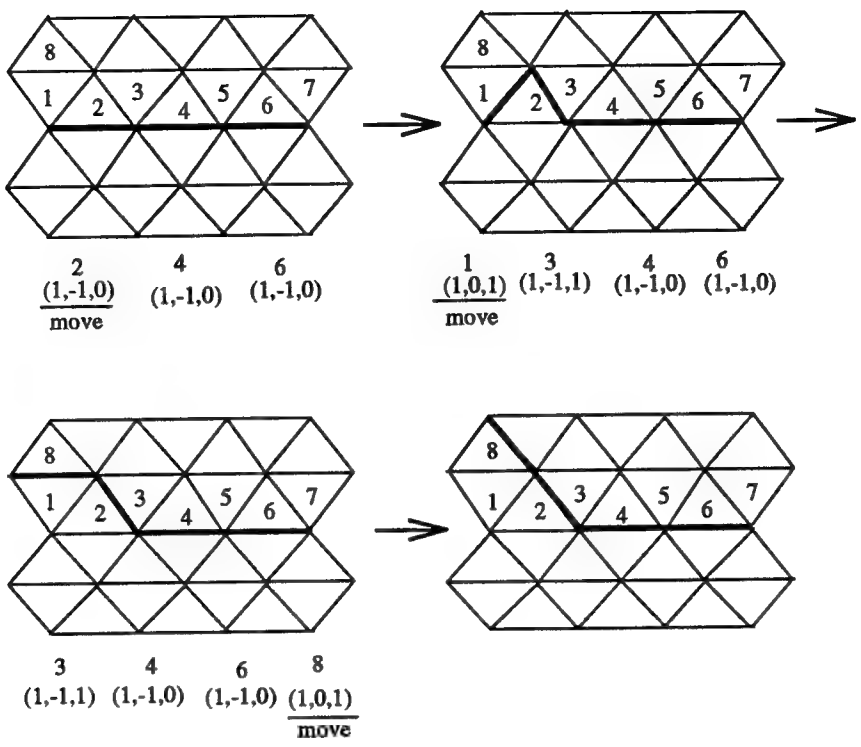

Figure 8: *Example showing how the cost template is used to move three elements.*

$$to\_send_{tot} = L_0 - \frac{\sum_{i=0}^{m} L_i}{m+1}.$$

The parent determines the individual amounts to_send_i to transfer to children in proportion to the their load request:

$$to\_send_i = min\{r_i, to\_send_{tot} \cdot \frac{r_i}{\sum_{j=1}^{m} r_j}\}.$$

The minimum of the two values is taken in order to prevent transferring loads greater than the requested load. Figure 9(b) shows the load requests and load grants for a subtree in the redistribution example.

Two important issues that should be addressed in an iterative load balancing algorithm are:

- *Convergence:* As the load is transferred iteratively between the processors, the load value on each processor should converge to the balanced average load in a finite number of steps.
- *Oscillations:* There should not be any indefinite cycles (repeating load transfer patterns) while the system is imbalanced.

Leiss/Reddy [9] prove the following results in [9]. Let an $H-$*neighborhood* denote the neighbors of a processor within a distance of H, C denote the load treshold value. If elements are taken as load units, then $C = 1$. Also let d indicate the *diameter* (the maximum of all the shortest paths

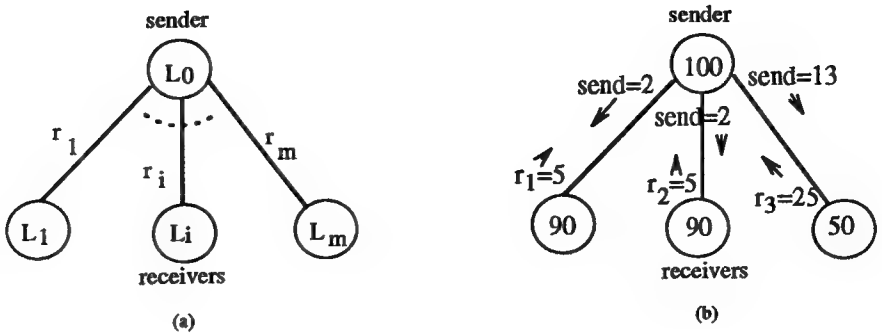

Figure 9: *(a) Load request* $r_i = \lceil (L_0 - L_i)/2 \rceil$ *from sender (b) example of transferred amounts*

between any two nodes) of the processor graph which in the present case is the partition graph. Finally, define H-neighborhood imbalance at time t as the variance

$$GIMB_H^t = \sum_{p \in P} (L(p) - \alpha)^2.$$

Here P is the processor set, $L(p)$, the load value on each processor and α the average load value per processor. The results proved are:

1. after a rebalancing iteration $GIMB_H^{t+1} < GIMB_H^t$ and
2. after balancing terminates, the maximum imbalance in the whole system is bounded by: $\lceil \frac{d}{2H} \cdot C \rceil$.

According to the first result, the imbalance in the *neighborhood* and *not* necessarily the whole system will reach a minimum since the $GIMB$ is decreased after each iteration. The second result states that if the system is neighborhood-balanced, the whole system can still be severely imbalanced. An example illustrating this worst case scenario is the configuration with n processors forming a 1 dimensional chain and each having a load that differs from neighbor only by $L_{i+1} - L_i = 1$, i.e. a load ramp. If $H = 1$ and $C = 1$ and since $d = n$, then the imbalance after termination of the algorithm will be $n/2$. Increasing the neighborhood measure H to $n/2$ will balance the system globally. However, $H = n/2$ will require *each* of the n processors to send messages to the $n/2$ H-neighbors. Hence choosing $H = n/2$ is impractical. In general, the case $H > 1$ will increase the communication volume and hence make the iterative balancing algorithm inefficient.

To avoid this problem with the Leiss/Reddy approach while keeping $H = 1$, two modifications are made to handle the case when the load difference between the neighboring processors is C. Unlike [4] which sends at most $\lfloor (L_0 - L_i)/2 \rfloor$ and considers the $L_0 - L_i = 1$ as balanced, the current procedure exchanges the excess load as given in Eq (2) even if the $GIMB$ will remain the same. Hence, it allows the case when $GIMB_H^{t+1} \leq GIMB_H^t$. The second modification involves, storing the previous move and making sure that the same excess load will not be transferred back to its original holder. This is performed in order to prevent oscillations between two neighboring processors i.e. oscillation on a cycle of length 2. Even though the first modification fixes the problem of premature termination without global load balancing, the second modification does not take care of detecting oscillations which can occur in cycles of length greater than 2.

6 Results

The parallel refinement and the redistribution algorithm has been tested on three problems. follows. Starting with an initial coarse mesh, partitioning was performed using the ORB method and the partitions mapped onto the torus connected MasPar MP-1 system. The mapped mesh was then refined selectively. Unstructured meshes on a square and a curved domain were used for testing the redistribution and refinement procedures. The square mesh was refined in one corner to simulate the difficult redistribution example involving a ramp load distribution. The curved domain was refined near the boundaries to test the parallel refinement and the redistribution algorithm over a highly unstructured mesh. Table 1 shows statistics for the test cases. The load on each processor in these examples was taken as the number of elements. Figures 7-9 shows the test meshes together with the plot of maximum load versus the redistribution iterations.

In all the test cases, an optimal load balance was obtained with the difference between the maximum and the average load per processor in the balanced system less than two. Even though, the number of iterations to global load balance took longer as shown in the convergence plots, in all the test cases, there was a sharp drop in the imbalance by the end of the first 10 iterations. Finally, it was noticed that the maximum number of partition boundary edges may increase as was the case in curved mesh.

Test	number of elements	Number of processors	Average elements per processor	Load (Min,Max) before	Load (Min,Max) after	Max boundary edges before	Max boundary edges after
square1	164	16	10.25	2, 32	7,11	14	12
square2	8296	1024	8.1	8,32	8,10	16	17
curved	1008	32	31.5	18,47	31,32	22	25

Table 1: Description the test cases.

7 Conclusion

This paper presented a procedure which performs parallel refinement and mesh redistribution to be used for adaptive finite element environments. The redistribution algorithm was based on the Leiss/Reddy heuristic and offered modifications to prevent possible premature termination. A new tree edge coloring algorithm was presented and used to synchronize the load transfers between processors. The whole system was tested on various meshes and showed good convergence results.

There are however some open problems in the load balancing scheme that has been used. First, the convergence *rate* for the Leiss/Reddy heuristic has not been established yet. This is particularly difficult to determine since the partition graph changes dynamically. Secondly, even though the modifications offered resolve oscillation on cycles of length 2, it is still not known how to efficiently detect oscillations on cycles of length greater than 2. Some state information has to be stored to detect such oscillations. The tests with a ramp load distribution which was vulnerable for oscillation did converge to the average load. However, this does not ensure that the problem cannot arise. Finally, the presented coloring algorithm makes it possible to use other element selection criteria to be used. In particular, coloring of processors enables the recursive subdivision techniques to be used locally on paired processors within the context of the Leiss/Reddy load balancing scheme.

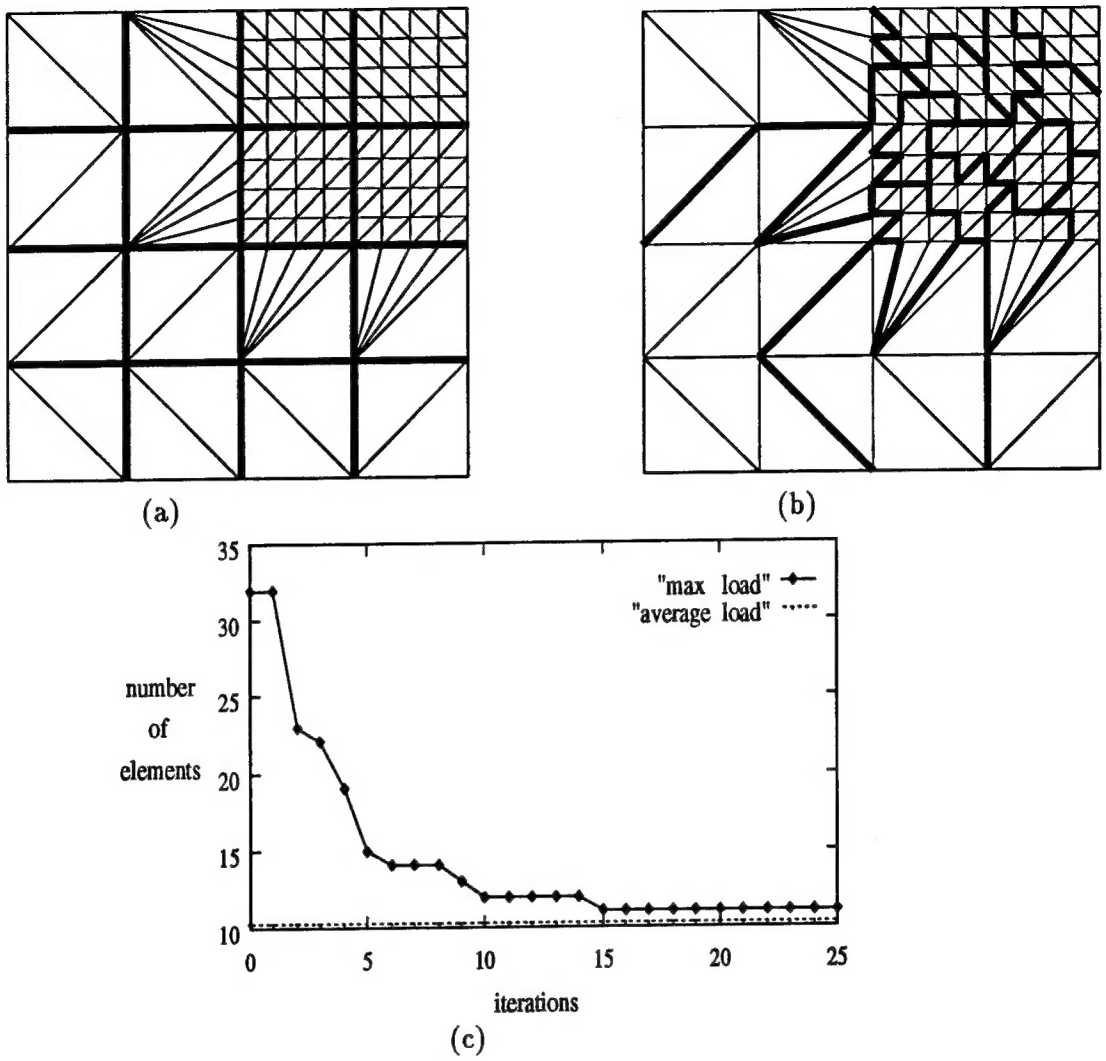

Figure 10: *Test square1: (a) unbalanced load after mesh refinement, (b) after redistribution and (c) convergence history.*

Acknowledgement

This research was supported by the U.S. Army Research Office under Contract Number DAAL-03-91-G-0215 and the National Science Foundation under Grant Number CCR 9216053.

References

[1] R. E. Bank and A. H. Sherman. An adaptive multi-level method for elliptic boundary value problems. *Computing*, 26:91–105, 1981.

[2] M. J. Berger and S. H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers*, C-36(5):570–580, May 1987.

[3] S.H. Bokhari, T.W. Crockett, and D. M. Nicol. Parametric binary dissection. Technical Report ICASE 93-39, ICASE, NASA Langley Res. Ctr.,Hampton, July 1993.

[4] K.D. Devine, J.E. Flaherty, S. R. Wheat, and A. B. Maccabe. A massively parallel adaptive finite element method with dynamic load balancing. Technical Report SAND 93-0936C, Sandia National Labs, Albuquerque, 1993.

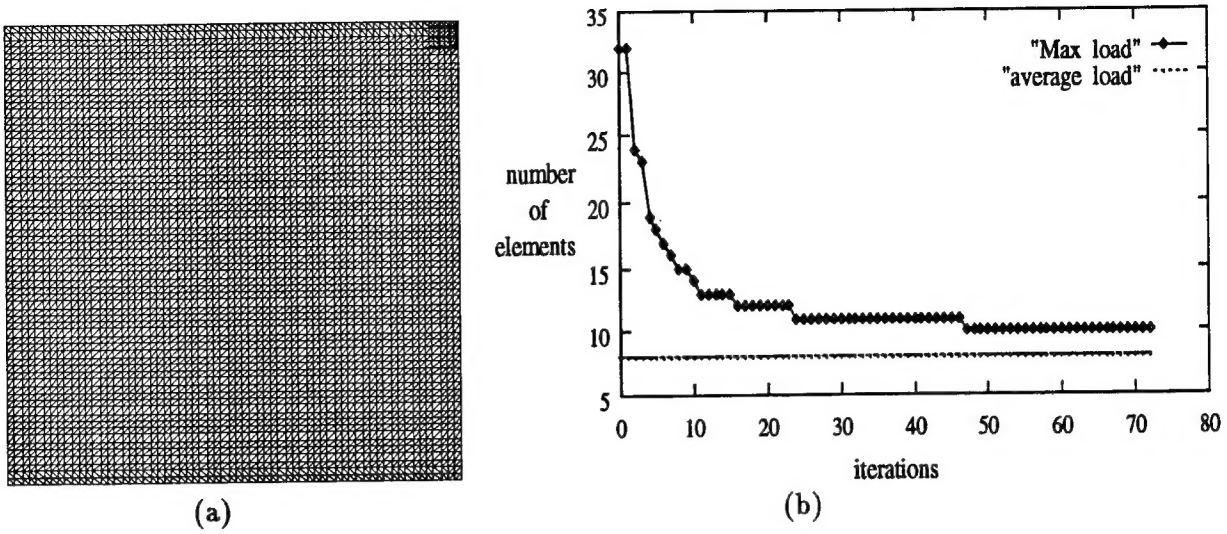

Figure 11: *Test square2: (a) square mesh on 32 by 32 processor array with refinement on a corner and (b) convergence history.*

[5] M. Fiedler. Algebraic connectivity of graphs. *Czechoslovak Math. J.*, 23:298–305, 1973.

[6] J.E. Flaherty, P.J. Paslow, M.S. Shephard, and J.D. Vasilakis, editors. *Adaptive Methods for Partial Differential Equations.* SIAM Proceedings Series, Philadelphia, 1989.

[7] S. W. Hammond. *Mapping Unstructured Grid Computations to Massively Parallel Computers.* PhD thesis, Computer Science Dept., Rensselaer Polytechnic Institue, Troy, 1991.

[8] J. JaJa. *An Introduction to Parallel Algorithms.* Addison-Wesley Publishing Company, Reading, 1992.

[9] E. Leiss and H. Reddy. Distributed load balancing: Design and performance analysis. *W. M. Keck Research Computation Laboratory*, 5:205–270, 1989.

[10] R. Lohner and R. Ramamurti. A parallelizable load balancing algorithm. In *Proc. of the AIAA 31st Aerospace Sciences Meeting and Exhibit*, Reno, 1993.

[11] A. Pothen, H. Simon, and K. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Anal. Appl.*, 11(3):430–452, July 1990.

[12] Maria-Cecilia Rivara. Algorithms for refining triangular grids suitable for adaptive and multigrid techniques. *International Journal for Numerical Methods in Engineering*, 20:745–756, 1984.

[13] M.S. Shephard and M.K Georges. Automatic three-dimensional mesh generation by the finite octree technique. *Int. J. Numer. Meth. Engng*, 32:709–749, 1991.

[14] B.A. Szabo and I. Babuska. *Introduction to Finite Element Analysis.* John Wiley, New York, 1991.

[15] V.G. Vizing. On an estimate of a chromatic class of a multigraph. In *Proc. Third Siberian Conf. on Mathematics and Mechanics*, Tomsk, 1964.

[16] C. Walshaw and M. Berzins. Dynamic load-balancing for pde solvers on adaptive unstructured meshes. Technical Report Preprint, School of Computer Studies, University of Leeds, Leeds, 1992.

[17] B. E. Webster, M. S. Shephard, and Z. Rusak. Unsteady compressible airfoil aerodynamics using an automated adaptive finite element method. *AHS Journal*, Submitted, 1993.

[18] K. J. Weiler. Edge based data structures for solid modeling in curved-surface environments. *IEEE Computer Graphics and Applications*, 5(2), 1985.

[19] S. R. Wheat, K. D. Devine, and A. B. Maccabe. Experience with automatic, dynamic load balancing and adaptive finite element computation. Technical Report SAND 93-2172A, Sandia National Labs, Albuquerque, 1993.

[20] R.D. Williams. Performance of dynamic load balancing algorithms for unstructured grid calculations. Technical Report C3P913, Pasadena, 1990.

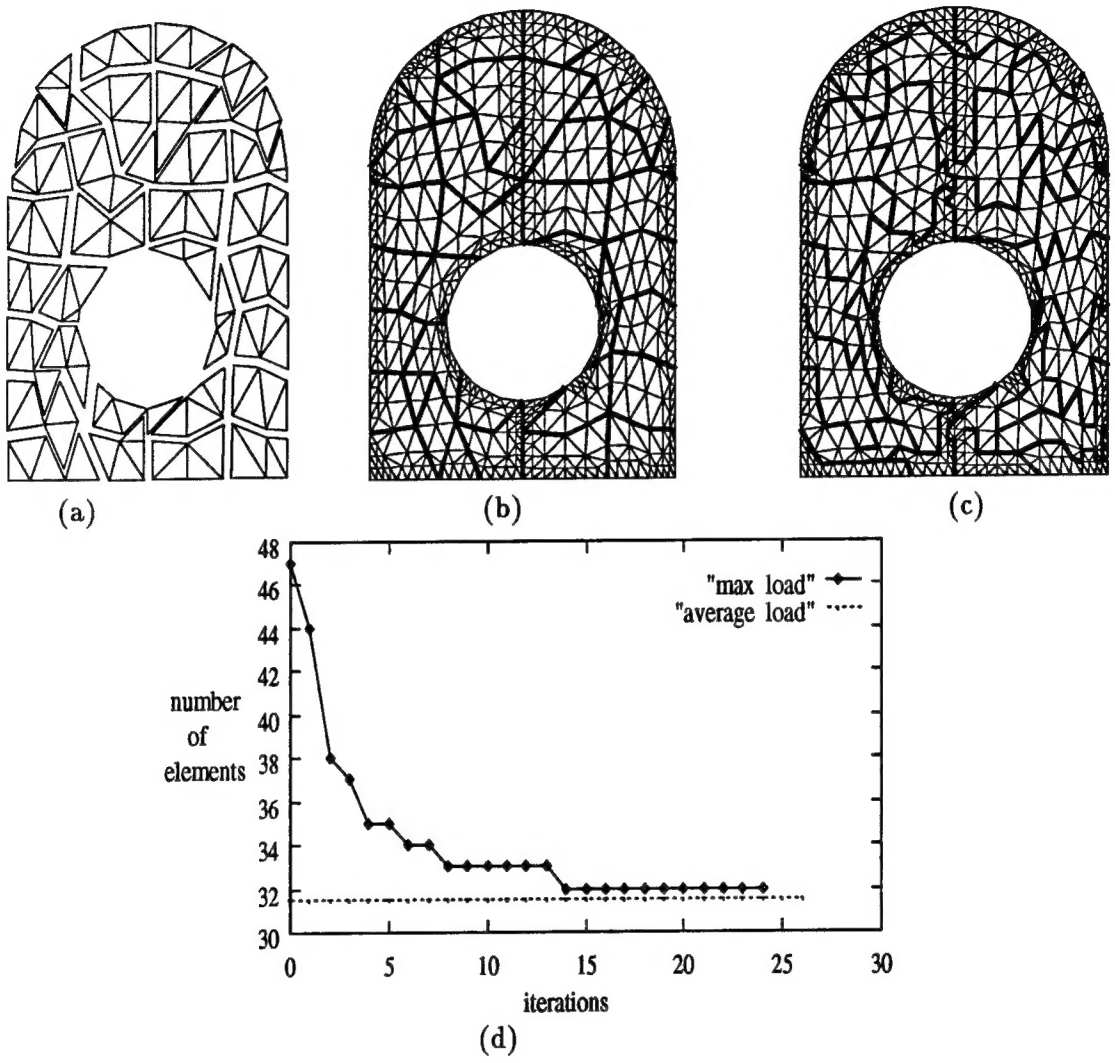

Figure 12: *Test curved (a) initial partitioned mesh after orthogonal recursive subdivision (b) refined mesh (c) redistributed mesh and (d) convergence history.*